Automatic Performance Programming?

\[ A \otimes I_n \]

Markus Püschel
Computer Science

\[
\begin{align*}
\_m128i \ t3 & = \_mm\_unpacklo\_epi16(X[0], X[1]); \\
\_m128i \ t4 & = \_mm\_unpackhi\_epi16(X[0], X[1]); \\
\_m128i \ t7 & = \_mm\_unpacklo\_epi16(X[2], X[3]); \\
\_m128i \ t8 & = \_mm\_unpackhi\_epi16(X[2], X[3]);
\end{align*}
\]
**Unlimited need for performance**

Many applications, but relatively few (~100 to 1000) components:

- Matrix multiplication
- Filters
- Fourier transform
- Coding/decoding
- Geometric transformations
- Graph algorithms
- ...

Fast components $\rightarrow$ fast applications
Software Performance: Traditional Approach

- Algorithms
- Software
- Compilers
- Microarchitecture

How well does that work?
The Problem: Example Fourier Transform

Discrete Fourier transform (single precision) on Intel Core i7 (4 cores)
Performance [Gflop/s]

- Fastest program (1 MB)
- Straightforward C (1 KB)

- Same opcount
- Best compiler
The Problem Is Everywhere

Matrix multiplication
Performance [Gflop/s]

WiFi Receiver
Performance [Mbit/s]

160x

30x
Model predictive control
Eigenvalues
LU factorization
Optimal binary search organization
Image color conversions
Image geometry transformations
Enclosing ball of points
Metropolis algorithm, Monte Carlo
Seam carving
SURF feature detection
Submodular function optimization
Graph cuts, Edmond-Karps Algorithm
Gaussian filter
Black Scholes option pricing
Disparity map refinement

Singular-value decomposition
Mean shift algorithm for segmentation
Stencil computations
Displacement based algorithms
Motion estimation
Multiresolution classifier
Kalman filter
Object detection
IIR filters
Arithmetic for large numbers
Optimal binary search organization
Software defined radio
Shortest path problem
Feature set for biomedical imaging
Biometrics identification
“Theorem:”

Let \( f \) be a mathematical function to be implemented on a state-of-the-art processor. Then

\[
\frac{\text{Performance of optimal implementation of } f}{\text{Performance of straightforward implementation of } f} \approx 10^{-100}
\]
Evolution of Processors (Intel)

Floating point peak performance [Gflop/s]
CPU frequency [GHz]

1993 1995 1997 1999 2001 2003 2005 2007 2009

Pentium
Pentium Pro
Pentium II
Pentium III
Pentium 4
Core 2 Duo
Core i7

- Orange circles: single precision
- Red circles: double precision
- Black squares: CPU frequency

free speedup
The End of Automatic Speedup

Floating point peak performance [Gflop/s]
CPU frequency [GHz]

100
10
1
0.1

work required

free speedup

1993 1995 1997 1999 2001 2003 2005 2007 2009

Pentium Pro
Pentium II
Pentium III
Pentium 4
Core 2 Duo
Core i7

single precision
double precision
CPU frequency

Era of parallelism
And There Will Be Variety …

Core i7

Arm Cortex A9

Nvidia G200

TI TNETV3020

Tilera Tile64

DFT: Analysis

Discrete Fourier transform (single precision) on Intel Core i7 (4 cores)
Performance [Gflop/s]

- Compiler doesn’t do the job (why?)
- Doing it by hand: requires guru knowledge
Optimization for Register Locality and ILP

// straightforward code
for (i = 0; i < N; i += 1) {
    for (j = 0; j < N; j += 1) {
        for (k = 0; k < N; k += 1) {
            c[i][j] += a[i][k]*b[k][j];
        }
    }
}

// unrolling + scalar replacement
for (i = 0; i < N; i += MU) {
    for (j = 0; j < N; j += NU) {
        for (k = 0; k < N; k += KU) {
            t1 = A[i*N + k];
            t2 = A[i*N + k + 1];
            t3 = A[i*N + k + 2];
            t4 = A[i*N + k + 3];
            t5 = A[(i + 1)*N + k];
            <more copies>
            t10 = t1 * t9;
            t17 = t17 + t10;
            t21 = t1 * t8;
            t18 = t18 + t21;
            t12 = t5 * t9;
            t19 = t19 + t12;
            t13 = t5 * t8;
            t20 = t20 + t13;
            <more ops>
            C[i*N + j] = t17;
            C[i*N + j + 1] = t18;
            C[(i+1)*N + j] = t19;
            C[(i+1)*N + j + 1] = t20;
    }
}

Removes aliasing
Enables register allocation and instruction scheduling

Compiler typically does not do:
• often illegal
• many choices
Optimization for Cache Locality

Many algorithms are formulated iteratively:
- many passes through data
- poor locality
- poor performance

Restructured for locality

*Compiler usually does not do*
- analysis may be unfeasible
- possibly many choices
- may require algorithm changes
- may require domain knowledge
- may require cache parameters
Optimization for Parallelism (Threads)

Restructured for locality

Parallelism is present, but is not in the “right shape”

Restructured for locality and parallelism (shared memory, 2 cores, 2 elements per cache line)

Compiler usually does not do

• analysis may be unfeasible
• may require algorithm changes
• may require domain knowledge
• may require processor parameters
Software Performance: Facts

- Straightforward code is often slow
- Inherent compiler limitations
- End of free speedup for legacy code
- Performance optimization is very hard
- Fast code violates good software engineering practices
- Performance optimization is “vertical”
  - algorithm changes
  - code style
  - considers microarchitectural parameters
- Highest performance is generally non-portable
Current practice: Thousands of programmers re-implement and re-optimize the same functionality for every new processor and for every new processor generation.
optimal op count

performance optimization
Performance is different than other software quality features

We need

- Courses on software performance
- Microarchitecture-cognizant algorithm analysis (e.g., Kung’s work, cache oblivious algorithms, …)
- Smarter compilers (e.g., iterative compilation, machine learning)
- “Vertical” software engineering and programming languages techniques for performance optimization
Organization

- Software performance issues
- Automatic performance tuning (autotuning)
- Automatic performance programming
- Conclusions
ATLAS/PhiPAC: MMM Generator
Bilmes et al. 97, Whaley et al. 98

// MMM loop-nest
for i = 0:N_B:N-1
    for j = 0:N_B:M-1
        for k = 0:N_B:K-1

// mini-MMM loop nest
for i’ = i:M_U:i+N_B-1
    for j’ = j:N_U:j+N_B-1
        for k’ = k:K_U:k+N_B-1

// micro-MMM loop nest
for k” = k’:1:k’+K_U-1
    for i” = i’:1:i’+M_U-1
        for j” = j’:1:j’+N_U-1

• ijk or jik depending on N and M
• Blocking for cache

Search parameters: N_B, M_U, N_U, K_U, L_S, ...

Offline tuning: tuning at installation
**FFTWin: Adaptive DFT Library**

*Frigo, Johnson 98, 05*

**Installation**

configure/make

**Usage**

\[ d = \text{dft}(n) \]

\[ d(x,y) \]

Search for fastest computation strategy

Online tuning: tuning at time of use
Autotuning

- **Use of models**
  - OSKI (Vuduc, Demmel, Yelick et al.)
  - Adaptive sorting (Li, Padua et al.)

- **Tools**
  - Iterative compilation (Knijnenburg, O’Boyle, Cooper, ...)
  - Machine learning in compilation (O’Boyle, Cavazos, ...)
  - Source code tools (POET, Rose)

*Most common form of autotuning:* Search over alternative implementations to find the fastest for a given platform

*Problem: usually parameter based*
- not portable to new types of platforms (e.g., parallel)
- not portable to different functions
Organization

- Software performance issues
- Automatic performance tuning (autotuning)
- *Automatic performance programming*
- Conclusions
Goal:

Computer generation of high performance code for ubiquitous performance-critical components
Possible Approach:

Capturing algorithm knowledge:  
*Domain-specific languages (DSLs)*

Structural optimization:  
*Rewriting systems*

High performance code style:  
*Compiler*

Decision making for choices:  
*Machine learning*
Spiral: Program Generation for Performance (www.spiral.net)
Linear Transforms

\[
\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} = y = Tx 
\]

Example:

\[
T = \text{DFT}_n = \left[ e^{-2\pi k\ell i/n} \right]_{0 \leq k, \ell < n}
\]
Algorithms: Example FFT, n = 4

Fast Fourier transform (FFT):

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & i & -1 & -i \\
1 & -1 & 1 & -1 \\
1 & -i & -1 & i \\
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
\end{bmatrix}
= 
\begin{bmatrix}
1 & . & . & 1 \\
. & 1 & . & 1 \\
1 & . & -1 & . \\
. & 1 & . & -1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 1 & . & . \\
. & 1 & . & 1 \\
1 & -1 & . & . \\
. & . & 1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & . \\
. & 1 \\
1 & . \\
. & 1 \\
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
\end{bmatrix}
\]

Data flow graph

Description with matrix algebra (SPL)

\[
\text{DFT}_4 = (\text{DFT}_2 \otimes I_2) T^4_2 (I_2 \otimes \text{DFT}_2) L^4_2
\]
Decomposition Rules (>200 for >40 Transforms)

Rules = algorithm knowledge

(≈100 journal papers)
# SPL to Code

<table>
<thead>
<tr>
<th>SPL $S$</th>
<th>Pseudo code for $y = Sx$</th>
</tr>
</thead>
</table>
| $A_nB_n$ | `<code for: t = Bx>
<code for: y = At>` |
| $I_m \otimes A_n$ | for (i=0; i<m; i++)
<code for:
y[i*n:1:i*n+n-1] = A(x[i*n:1:i*n+n-1])> |
| $A_m \otimes I_n$ | for (i=0; i<n; i++)
<code for:
y[i:n:i+m*n-n] = A(x[i:n:i+m*n-n])> |
| $D_n$ | for (i=0; i<n; i++)
y[i] = D[i]*x[i]; |
| $L^k_m$ | for (i=0; i<k; i++)
for (j=0; j<m; j++)
y[i*m+j] = x[j*k+i]; |
| $F_2$ | y[0] = x[0] + x[1];
y[1] = x[0] - x[1]; |

$\mathbf{I_m \otimes A_n} = \begin{bmatrix} A_n & \cdots \\
\vdots & \\
& A_n \end{bmatrix}$

*Gives reasonable, straightforward code*
Program Generation in Spiral

<table>
<thead>
<tr>
<th>Transform</th>
<th>DFT₈</th>
</tr>
</thead>
</table>

Decomposition rules (algorithm knowledge)

<table>
<thead>
<tr>
<th>Algorithm (SPL)</th>
<th>(DFT₂ ⊗ I₄) T₄⁸ (I₂ ⊗ ((DFT₂ ⊗ I₂) T₂⁴ (I₂ ⊗ DFT₂) L₄) L₂)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Algorithm (Σ-SPL)</th>
<th>∑ (Sₐ DFT₂ Gₐ) ∑ ( ∑ (Sₖ,l diag(tₖ,l) DFT₂ Gₙ) ∑ (Sₘ diag(tₘ) DFT₂ Gₘ,k,m))</th>
</tr>
</thead>
</table>

C Program

```c
void sub(double *y, double *x) {
    double f0, f1, f2, f3, f4, f7, f8, f10, f11;
    f0 = x[0] - x[3];
    f1 = x[0] + x[3];
    f2 = x[1] - x[2];
    f3 = x[1] + x[2];
    f4 = f1 - f3;
    f7 = 0.7071067811865476 * f4;
    f8 = 0.9238795325112867 * f0;
    y[0] = f1 + f3;
    y[2] = 0.7071067811865476 * f4;
    y[3] = 0.9238795325112867 * f0;
< more lines>
```
SPL to Shared Memory Code: Basic Idea

“Good” SPL structures

\[ y = \left( I_p \otimes A \right) x \]

\[ y = \left( P \otimes I_\mu \right) x \]

Rewriting: Bad structures \(\rightarrow\) good structures
Example: SMP Parallelization

\[
\overbrace{\text{DFT}_{mn}}^{\text{sm}(p,\mu)} \quad \rightarrow \quad \left( \frac{\left( \text{DFT}_m \otimes I_n \right) \tau_{mn}^{\text{sm}(p,\mu)} \left( I_m \otimes \text{DFT}_n \right) L_{mn}^{\text{sm}(p,\mu)} \right) \right.
\]

\[
\vdots
\]

\[
\rightarrow \quad \left( \frac{\text{DFT}_m \otimes I_n}{\text{sm}(p,\mu)} \right) \left( \frac{\tau_{mn}^{\text{sm}(p,\mu)}}{\text{sm}(p,\mu)} \right) \left( \frac{I_m \otimes \text{DFT}_n}{\text{sm}(p,\mu)} \right) \left( \frac{L_{mn}^{\text{sm}(p,\mu)}}{\text{sm}(p,\mu)} \right)
\]

\[
\vdots
\]

\[
\rightarrow \quad \left( \frac{\left( L_{mn}^{mp} \otimes I_{n/p,\mu} \right) \otimes I_{\mu}}{p-1 \sum_{i=0}^{p-1} \tau_{mn,i}^{\text{sm}(p,\mu)}} \right) \left( I_p \otimes \left( I_{m/p} \otimes \text{DFT}_n \right) \right) \left( I_p \otimes L_{mn/p}^{mp} \right) \left( L_{m/p}^{pm} \otimes I_{m/p,\mu} \right) \otimes I_{\mu}
\]

load-balanced, no false sharing

One rewriting system for every platform paradigm: SIMD, distributed memory parallelism, FPGA, …
Challenge: General Size Libraries

So far:

*Code specialized to fixed input size*

\[
\text{DFT}_384(x, y) \{
    \ldots
    \text{for}(i = \ldots) \{ \\
        t[2i] = x[2i] + x[2i+1] \\
        t[2i+1] = x[2i] - x[2i+1] \\
    \}
    \ldots
\}
\]

- Algorithm fixed (offline tuning)
- Nonrecursive code

Challenge:

*Library for general input size*

\[
\text{DFT}(n, x, y) \{
    \ldots
    \text{for}(i = \ldots) \{ \\
        \text{DFT}_\text{strided}(m, x+mi, y+i, l, k) \\
    \}
    \ldots
\}
\]

- Algorithm cannot be fixed (online tuning)
- Recursive code
- Creates many challenges
Challenge: Recursive Composition

16 = 4 × 4

void dft(int n, cpx *y, cpx *x) {

(DFT_k ⊗ I_m) T_{km}^{km} (I_k ⊗ DFT_m) L_{km}^{km}

...
Σ–SPL : Basic Idea

Four additional matrix constructs: Σ, G, S, Perm

- Σ (sum) explicit loop
- G_f (gather) load data with index mapping \( f \)
- S_f (scatter) store data with index mapping \( f \)
- Perm_f permute data with the index mapping \( f \)

Example: \[ y = (I_4 \otimes F_2)x \rightarrow y = \sum_{j=0}^{3} S_{f_j} F_2 G_{f_j} x \]
Find Recursion Step Closure
Voronenko, 2008

\[
\begin{align*}
\{\text{DFT}_n\} \\
(\{\text{DFT}_{n/k}\} \otimes I_k)T_k^n(I_{n/k} \otimes \{\text{DFT}_k\})L^n_{n/k} \\
(k-1) \sum_{i=0}^{k-1} S_{h_i,k} \{\text{DFT}_{n/k}\} G_{h_i,k} \ \text{diag}(f) \sum_{j=0}^{n/k-1} S_{h_{jk},1} \{\text{DFT}_k\} G_{h_{jk},1} \ \text{perm}(\ell^n_{n/k}) \\
(k-1) \sum_{i=0}^{k-1} S_{h_i,k} \{\text{DFT}_{n/k}\} \text{diag}(f \circ h_i,k) G_{h_i,k} \sum_{j=0}^{n/k-1} S_{h_{jk},1} \{\text{DFT}_k\} G_{h_j,n/k} \\
k-1 \sum_{i=0}^{k-1} \{S_{h_i,k} \ \text{DFT}_{n/k}\} \text{diag}(f \circ h_i,k) G_{h_i,k} \sum_{j=0}^{n/k-1} \{S_{h_{jk},1} \ \text{DFT}_k\} G_{h_j,n/k}
\end{align*}
\]

Repeat until closure
Recursion Step Closure: Examples

**DFT: scalar code**

**DFT: full-fledged (vectorized and parallel code)**
It Really Works

DFT on Sandybridge (3.3 GHz, 4 Cores, AVX)

Performance [Gflop/s]

- Many transforms, often the generated code is best
- Vector, shared/distributed memory parallel, FPGAs

---

DFT_n → (DFT_k ⊗ I_m)T^m_n(I_k ⊗ DFT_m)L^k_n
DFT_n → P^T_{k/2,m} DFT_m ⊗ (I_{k/2 - 1} ⊗ C_{2,m} rDFT_2m(i/k)) (RDFT_k ⊗ I_m)
RDFT_n → (P^T_{k/2,m} ⊗ I_2) (RDFT_2m ⊗ (I_{k/2 - 1} ⊗ D_{2,m} rDFT_2m(i/k))) (RDFT_k ⊗ I_m)
rDFT_2m(u) → L^m_{2m}(I_k ⊗ 1 rDFT_2m((i + u)/k)) (rDFT_k(u) ⊗ I_m)

5MB vectorized, threaded, general-size, adaptive library
Computer generated Functions for Intel IPP 6.0

3984 C functions
1M lines of code

Transforms: DFT (fwd+inv), RDFT (fwd+inv), DCT2, DCT3, DCT4, DHT, WHT
Sizes: 2–64 (DFT, RDFT, DHT); 2-powers (DCTs, WHT)
Precision: single, double
Data type: scalar, SSE, AVX (DFT, DCT), LRB (DFT)

Computer generated

Results: SpiralGen Inc.
### Online tuning

**Installation**
configure/make

**Use**
\[ d = \text{dft}(n) \]
\[ d(x,y) \]

Search for fastest computation strategy

### Offline tuning

**Installation**
configure/make

for a few \( n \): search
learn decision trees

**Use**
\[ d = \text{dft}(n) \]
\[ d(x,y) \]

### Machine learning

```c
if ( n <= 65536 ) {
    if ( n <= 32 ) {
        if ( n <= 4 ) { return 2; }
        else { return 4; }
    }
    else { return 4; }
}
else {
    if ( n <= 1024 ) {
        if ( n <= 256 ) { return 8; }
        else { return 32; }
    }
    else {
        // OpenMP loop of scaled dfts
    }
```
Program Generators: Related Work

- FFTW codelet generator (Frigo)
- Flame (van de Geijn, Quintana-Orti, Bientinesi, ...)
- Tensor contraction engine (Baumgartner, Sadayappan, Ramanujan, ...)
- cvxgen (Mattingley, Boyd)
- PetaBricks (Ansel, Amarasinghe, ...)
- Spiral
Organization

- Software performance issues
- Automatic performance tuning (autotuning)
- Automatic performance programming
- Conclusions
- End of free speedup
- Straightforward code often 10-100x suboptimal
- Performance ≠ efficient parallelization
- Likely inherent compiler limitations
- Performance optimization violates good programming practices
So What Do We Do?

- Teaching
- Better autotuning

*Search over parameterized alternatives*

Automating performance optimization with tools that complement/aid the compiler or programmer.

Example: Program generation for performance

*Maybe Spiral can be generalized*

- DSLs
- Rewriting for platform paradigms
- Search/learning